1. 1、介绍一下join操作优化经验?

答：join其实常见的就分为两类： map-side join 和 reduce-side join。当大表和小表join时， 用map-side join能显著提高效率。将多份数据进行关联是数据处理过程中非常普遍的用法，不过 在分布式计算系统中，这个问题往往会变的非常麻烦，因为框架提供的 join 操作一般会将所有数 据根据 key 发送到所有的 reduce 分区中去，也就是 shuffle 的过程。造成大量的网络以及磁盘 IO消耗，运行效率极其低下，这个过程一般被称为 reduce-side-join。如果其中有张表较小的 话，我们则可以自己实现在 map 端实现数据关联，跳过大量数据进行 shuffle 的过程，运行时间 得到大量缩短，根据不同数据可能会有几倍到数十倍的性能提升。 （备注：这个题目面试中非常非常大概率见到，务必搜索相关资料掌握，这里抛砖引玉。）

2. 2、spark.storage.memoryFraction参数的含义,实际生产中如何调优?

答：1）用于设置RDD持久化数据在Executor内存中能占的比例，默认是0.6,，默认Executor 60%的内存，可以用来保存持久化的RDD数据。根据你选择的不同的持久化策略，如果内存不够 时，可能数据就不会持久化，或者数据会写入磁盘。
2）如果持久化操作比较多，可以提高 spark.storage.memoryFraction参数，使得更多的持久化数据保存在内存中，提高数据的读取性 能，如果shuffle的操作比较多，有很多的数据读写操作到JVM中，那么应该调小一点，节约出更 多的内存给JVM，避免过多的JVM gc发生。在web ui中观察如果发现gc时间很长，可以设置 spark.storage.memoryFraction更小一点。

3. 3、spark.shuffle.memoryFraction参数的含义，以及优化经验?

答：1）spark.shuffle.memoryFraction是shuffle调优中 重要参数，shuffle从上一个task拉去数 据过来，要在Executor进行聚合操作，聚合操作时使用Executor内存的比例由该参数决定，默认 是20% 如果聚合时数据超过了该大小，那么就会spill到磁盘，极大降低性能；
2）如果Spark作业中的 RDD持久化操作较少，shuffle操作较多时，建议降低持久化操作的内存占比，提高shuffle操作的 内存占比比例，避免shuffle过程中数据过多时内存不够用，必须溢写到磁盘上，降低了性能。此 外，如果发现作业由于频繁的gc导致运行缓慢，意味着task执行用户代码的内存不够用，那么同 样建议调低这个参数的值。

4. 4、Spark性能优化主要有哪些手段?

判断内存消耗：设置RDD并行度，将RDD cache到内存，通过BlockManagerMasterActor添加RDD到memory中的日志查看每个partition占了多少内存，最后乘以partition数量，即是RDD内存占用量。
1.Shuffle调优（核心）
a.consolidation机制，使shuffleMapTask端写磁盘的文件数量减少，resultTask拉取数据磁盘IO也变少，只需拉取cpu core数量的磁盘文件。
b.spark.shuffle.file.buffer:shuffleMapTask的写磁盘bucket缓存，默认32K，加大后溢出到磁盘的次数变少。
c.spark.reducer.maxSizeInFlight:reduce task的拉取缓存默认48M，可以加大缓存，拉取次数减少。
d.reduce task拉取数据时，map task的jvm正在full gc（时间过长导致数据丢失），工作线程暂停，数据拉取不到，设置重试次数和间隔。
e.reduce task的executor有一部分内存用来汇聚拉取的数据，放入map。调整比例，防止频繁溢写到

磁盘。
2.对多次使用的RDD进行持久化/checkpoint
a.通过cache和persist将RDD数据持久化存储到BlockManager，如果数据丢失，第二次计算该RDD时，会先尝试读取checkPoint数据，读取不出来就只能重新计算RDD。（cache只有一个默认的缓存级别MEMORY_ONLY，而persist可以根据情况设置其他的缓存级别）
b.要求高性能就在第一次计算RDD后，对RDD进行checkpoint操作。
3.使用序列化的持久化级别
a.将RDD序列化之后，RDD的每个partition的数据就序列化为一个巨大的字节数组，再持久化，可以大大减小对内存的消耗，同时数据量小了之后，如果要写入磁盘，那么磁盘IO性能消耗也比较小。缺点是反序列化时会增大cpu性能开销。
4.提高并行度
a.推荐设置集群总CPU数量的两倍~三倍的并行度，这样每个cpu core可能分配到并发运行2~3个task线程。那么集群资源就不会空闲，连续运转发挥最大功效。（spark.default.parallelism设置统一的并行度）

5. 5.数据本地化

a.PROCESS_LOCAL，进程本地化，rdd的partition和task进入一个Executor中，速度快。
b.NODE_LOCAL，rdd的partition和task，不在一个executor，不在一个进程，在一个worker。
c.NO_PREF，无所谓本地化级别，数据在哪性能都一样。
d.RACK_LOCAL，机架本地化。
e.ANY，任意的本地化级别。
spark.locality.wait、spark.locality.wait.nade、spark.locality.waitprocess等参数设置数据本地化等待时间

6. 6.使用高性能序列化类库

a.默认使用java序列化机制，速度比较慢，占用内存空间大
b.kryo序列化速度快，占用内存空间小，但是并不一定对所有类型进行序列化。需要对序列化的自定义类型进行注册（避免保存对象全类名），根据对象大小优化缓存大小。

7. 7.优化数据结构

a.优先使用数组以及字符串，少用集合类。String拼接成特殊格式的字符串。如id:name，address|id:name，address。
b.避免使用多层嵌套的对象结构。可以采用json字符串。
c.尽量使用int替代String。id不要用uuid，用自增的int类型id。

8. 8.广播共享数据

a.默认情况算子函数使用到的外部数据，会被拷贝到每一个task中。
b.通过使用外部大数据进行Broadcast广播，让其在每一个节点上就一份副本，而不是每个task一份副本，减少内存占用空间和网络传输消耗。

9. 9.reduceBykey和groupBykey的合理使用

a.reduceBykey适合key对应的values聚合为一个值的场景，会先在shuffleMapTask写入本地磁盘文件进行本地聚合，导致传输到resultTask时网络传输数据量减小。
b.groupByKey不会进行本地聚合，把ShuffleMapTask的输出拉取到ResultTask内存中，网络传输开销大。
5、spark 如何防止内存溢出？
1). map过程产生大量对象导致内存溢出
这种溢出的原因是在单个map中产生了大量的对象导致的。
例如：rdd.map(x=>for(i <- 1 to 10000) yield i.toString)，这个操作在rdd中，每个对象都产生了10000个对象，这肯定很容易产生内存溢出的问题。针对这种问题，在不增加内存的情况下，可以通过减少每个Task的大小，以便达到每个Task即使产生大量的对象Executor的内存也能够装得下。具体做法可以在会产生大量对象的map操作之前调用repartition方法，分区成更小的块传入map。例如：
rdd.repartition(10000).map(x=>for(i <- 1 to 10000) yield i.toString)。
面对这种问题注意，不能使用rdd.coalesce方法，这个方法只能减少分区，不能增加分区，不会有shuffle的过程。
2).数据不平衡导致内存溢出
数据不平衡除了有可能导致内存溢出外，也有可能导致性能的问题，解决方法和上面说的类似，就是调用repartition重新分区。这里就不再累赘了。
3).coalesce调用导致内存溢出
这是我最近才遇到的一个问题，因为hdfs中不适合存小问题，所以Spark计算后如果产生的文件太小，我们会调用coalesce合并文件再存入hdfs中。但是这会导致一个问题，例如在coalesce之前有100个文件，这也意味着能够有100个Task，现在调用coalesce(10)，最后只产生10个文件，因为coalesce并不是shuffle操作，这意味着coalesce并不是按照我原本想的那样先执行100个Task，再将Task的执行结果合并成10个，而是从头到位只有10个Task在执行，原本100个文件是分开执行的，现在每个Task同时一次读取10个文件，使用的内存是原来的10倍，这导致了OOM。解决这个问题的方法是令程序按照我们想的先执行100个Task再将结果合并成10个文件，这个问题同样可以通过repartition解决，调用repartition(10)，因为这就有一个shuffle的过程，shuffle前后是两个Stage，一个100个分区，一个是10个分区，就能按照我们的想法执行。
4.shuffle后内存溢出
shuffle内存溢出的情况可以说都是shuffle后，单个文件过大导致的。在Spark中，join，reduceByKey这一类型的过程，都会有shuffle的过程，在shuffle的使用，需要传入一个partitioner，大部分Spark中的shuffle操作，默认的partitioner都是HashPatitioner，默认值是父RDD中最大的分区数,这个参数通过spark.default.parallelism控制(在spark-sql中用spark.sql.shuffle.partitions)，
spark.default.parallelism参数只对HashPartitioner有效，所以如果是别的Partitioner或者自己实现的Partitioner就不能使用spark.default.parallelism这个参数来控制shuffle的并发量了。如果是别的partitioner导致的shuffle内存溢出，就需要从partitioner的代码增加partitions的数量。

5、standalone模式下资源分配不均匀导致内存溢出
在standalone的模式下如果配置了–total-executor-cores 和 –executor-memory 这两个参数，但是没有配置–executor-cores这个参数的话，就有可能导致，每个Executor的memory是一样的，但是cores的数量不同，那么在cores数量多的Executor中，由于能够同时执行多个Task，就容易导致内存溢出的情况。这种情况的解决方法就是同时配置–executor-cores或者spark.executor.cores参数，确保Executor资源分配均匀。

6、Sort-based shuffle的缺陷?
1）. 如果Mapper中Task的数量过大，依旧会产生很多小文件，此时在Shuffle传递数据的过程中到Reducer端，reduce会需要同时打开大量的记录来进行反序列化，导致大量的内存消耗和GC的巨大负担，造成系统缓慢甚至崩溃！
2）．如果需要在分片内也进行排序的话，此时需要进行Mapper端和Reducer端的两次排序！！！
优化：
可以改造Mapper和Reducer端，改框架来实现一次排序。
频繁GC的解决办法是：钨丝计划！！